OpenComRTOS: Reliable performance with a small code size

Bernhard H.C. Sputh[1], Oliver Faust[2], Eric Verhulst[2], Vitaliy Mezhuyev[1] and Tom Tierens[3]
[1]{bernhard.sputh, vitaliy.mezhuyev}@openlicensesociety.org
[2]{oliver.faust, eric}@altreonic.com [3]tom.tierens@denayer.wenk.be

Abstract

OpenComRTOS is one of the few Real-Time Operating Systems for embedded systems that was developed using formal modeling techniques. The goal was to obtain a proven trustworthy component with a clean architecture which delivers high performance on a wide variety of networked embedded systems, ranging from single processor to distributed systems. The result is a scalable communication system with real-time capabilities. Besides, the rigorous formal verification of the kernel algorithms lead to an architecture which has several properties that enhance safety and real-time properties of the RTOS. The code size in particular is very small, typically 10 times less compared with a typical equivalent single processor RTOS. The latter allows a much better use of the on-chip memory resources.

To this point we ported OpenComRTOS to the MicroBlaze processor from Xilinx. In this processor environment OpenComRTOS competes with a number of different operating systems, including the standard operating system Xilinx Micro Kernel. This paper reports code size figures of the OpenComRTOS on a MicroBlaze target. We found that this code size is considerably smaller compared with published code sizes of other operating systems.

1. Introduction

Real-Time Operating Systems (RTOSs) are a key software module for embedded systems, often requiring properties of high reliability and safety. Unfortunately, most commercial, as well as open source implementations cannot be verified or even certified, e.g. according to the DoD_178B or IEC61508 standards. Similarly, software engineering is often done in a non-systematic way, although well defined and established Systems Engineering Processes exist [1, 6]. The software is rarely proven to be correct even though this is possible with formal model checkers [5]. In the context of a unified systems engineering approach [2] we undertook a research project were we followed a stricter methodology, including formal model checking, to obtain a network-centric RTOS which can be used as a trusted component.

1.1 General requirements for OpenComRTOS

The history for this project goes back to the early 1990's when a distributed real-time RTOS called Virtuoso (Eonic Systems) was developed for the INMOS transputer [10]. This processor had build in support for concurrency as well as interprocess communication and was enabled for parallel processing by way of 4 communication links. Virtuoso allowed such a network of processors to be programmed in a topology transparent way. Later, the software evolved and was ported from single chip microcontrollers to systems with over a thousand Digital Signal Processors until the technology was acquired by Wind River and after a few years they removed it from the market. The OpenComRTOS project was motivated by lessons learned from developing three Virtuoso generations. These lessons became part of the requirements. We list the most important ones:

- Scalability: The RTOS should support very small single processor systems, as well as widely distributed processing systems interconnected through external networks like the internet. To achieve that, the developing software must be independent of the mapping onto the network topology.
- Efficiency: The essence of multi-processor systems is communication. The challenge, from an RTOS point of view, is keeping the latency to a minimum while at the same time maximizing the performance. This is achieved when most of the critical code resides in the limited amount of on-chip memory.
- Small code size: This has a double benefit: a) performance and b) less complexity. Less complex systems have fewer potential sources of errors and side-effects.
- Trustworthy: As testing of distributed systems becomes very time consuming, it is mandatory that the

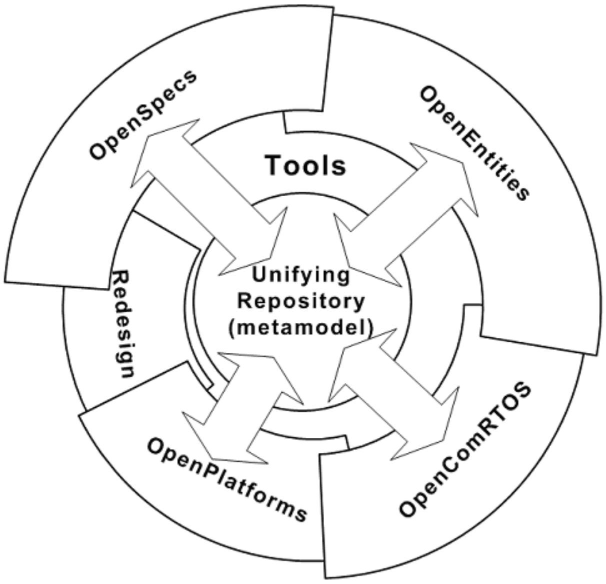

Figure 1. Open License Society: the unified view

system software can be trusted from the start. As errors typically occur in "corner cases", the use of formal methods was deemed necessary.

- Maintainability and ease of development: The code needs to be clear and simple and facilitate the development of e.g. drivers, the latter often been the weak point in system software.

OpenComRTOS provides the runtime environment supporting these requirements. The remainder of this paper focuses on this runtime environment and the execution on a MicroBlaze target. But, before we discuss the details of OpenComRTOS in greater detail, we deduce a two general points from the list of requirements.

The scalability requirement imposes that datacommunication is central in the RTOS architecture. Therefore, OpenComRTOS is network centric. The trustworthiness and maintainability aspects are addressed in the context of a Systems Engineering methodology, the use of common semantics during all activities is crucial. Because only common semantics enable us to generate most of the implementation code from the modeling and simulation phase. Generated code is more trustworthy compared with handwritten code. Using an "Interacting Entities" paradigm requires a runtime environment that supports concurrency and synchronization/communication in a native way between concurrent entities.

2. OpenComRTOS architecture

Even with the points mentioned above, Virtuoso was a successful product. The goal was to improve on its weaknesses. Its architecture was performant but very hard to port and to maintain. Hence, for OpenComRTOS we adopted a layered architecture which is based on semantic layering. The lowest functionality level (L0) is limited to priority based preemptive multitasking. In L0 Tasks exchange standardized Packets using an intermediate entity we called Ports. Hence, Tasks can synchronise and communicate using Packets and Ports. The Packets are the essential workhorse of the system. They have header and data fields and are exclusively used for all services, rather than invoking function calls or using jump tables. Hence, it becomes straightforward to provide services that operate in a transparent way across processor boundaries. Furthermore, Packets are very efficient, because kernel operations often come down to shuffling around Packets (using handlers) between system level datastructures.

At the next semantic level (L1) we added more traditional RTOS services like events, semaphores, queues, resources, etc. Finally, the architecture was kept simple and modular by developing kernel and drivers as Tasks. All these Tasks have a 'Task input Port' for accepting Packets from other Tasks. This has some unusual consequences like: a) the possibility to process interrupts received on one processor on another processor, b) the kernel having a lower priority than the drivers or even c) having multiple kernel Tasks on a single node.

2.1 Systems Engineering approach

The Systems Engineering approach from Open License Society, outlined in Figure 1, is a classical one as defined in [2], but adapted to the needs of embedded software development. It is first of all an evolutionary process using continuous iterations. In such a process, much attention is paid to an incremental development requiring regular review meetings by several of the stakeholders. On an architectural level, the system or product under development is defined under the paradigm of "Interacting Entities", which maps very well on an RTOS based runtime system. Applied to the development of OpenComRTOS, the process was started by elaborating a first set of requirements and specifications. Next, an initial architecture was defined. From this point on, two groups started to work in parallel. The first group worked out an architectural model, while a second group developed initial formal models using TLA+/TLC [9]. These models were incrementally refined.

Note that no real attempt was made to model the complete system at once. First of all, this is not possible in a generic way, because formal TLA models cannot be parameterised. For example, one must model a specific set of tasks and services this leads very quickly to a state explosion which limits the achievable complexity of such models. Hence, we modeled only specific parts, e.g. a model was build for each class of services (Ports, Events, Semaphores, etc.). This was sufficient and gives the benefit of having very clean, orthogonal models.

At each review meeting between the software engineers and the formal modeling engineer, more details were added to the models, the models were checked for correctness and a new iteration was started. This process was stopped when the formal models were deemed close enough to the implementation architecture. Next, a simulation model was developed on a PC (using Windows NT as a virtual target). This code was then ported to a real 16bit microcontroller [5]. On this target a few specific optimizations were performed on the implementation, while fully maintaining the design and architecture. The software was written in ANSI C and verified for safe coding practices with a MISRA rule checker [3].

2.2. Lessons from using formal modeling

The goal of using formal techniques is the ability to prove that the software is correct. This is an often heard statement from the formal techniques community. A first surprise was that each model gave no errors when verified by the TLC model checker. This is actually due to the iterative nature of the model development process and partly its strength. From an initial rather abstract model, successive models are developed by checking them using the model checker and hence each model is correct when the model checker finds no illegal states. As such, model checkers can't proof that the software is correct. They can only proof that the formal model is correct. For a complete proof of the software the whole programming chain as well as the target hardware should be modeled and verified. This is an unachievable goal due to its complexity and the resulting state space explosion. It was nevertheless attempted in the Verisoft project [4]. The model itself would be many times larger than the developed software. This indicates that if we would make use of verified target processors and verified programming language compilers, model checking becomes practical, because it is limited to modeling the application.

Other issues, related to formal modelling, were also discovered. A first issue is that the TLC model checker declares every action as a critical section, whereas e.g. in the case of a RTOS, many components operate concurrently and real-time performance dictates that on a real target the critical sections are kept as short as possible. This dictates us to avoid shared data structures, however it would be helpful to have formal model assistance that indicates the required critical sections.

2.3. Benefits obtained from using formal modeling

As was outlined above, the use of formal modeling was found to result in a much better architecture. This benefit results from successive iteration and review of the model. Another reason for the better architecture is the fact that formal model checkers provide a higher level of abstraction compared with the implementation. In the project we found that the semantics associated with specific programming terms involuntarily influence choices made by the architecting engineer. An example was the use of both waiting lists and Port buffers, which is one of the main concepts of OpenComRTOS. A waiting list is associated just with a waiting action but one overlooks that it also provides buffering behavior. Hence, one waiting list is sufficient resulting in a smaller and cleaner architecture.

Formal modeling and abstract levels have helped to introduce, define and maintain orthogonal architectural concepts. Orthogonality is key to small and safe, i.e. reliable, designs. Similarly, even if there was a short learning curve to master the mathematical notation in TLA, with hindsight this was an advantage vs. e.g. SPIN [8] that uses a C-like syntax. The latter leads automatically to thinking in terms of an implementation code with all its details, whereas the abstraction of TLA helps to think in more abstract terms. This also highlights the importance of specifying first before implementation is started.

A final observation is that using formal modeling techniques turned out to be a much more creative process than the mathematical framework suggests. TLA/TLC as such was primarily used as an architectural design tool, aiding the team in formulating ideas and testing them in a rather abstract way. This was proven to be a team work with lots of human interaction between the team members. The formal verification of the RTOS itself was basically a side-effect of building and running the models. Hence, this project has shown how a combination of team work with extensive peer-review, formal modeling support and a well defined goal can result in a "correct-by-design" product.

2.4. Novelties in the architecture

OpenComRTOS has a semantically layered architecture. At the lowest level (L0) the minimum set of Entities provides everything that is needed to build a small networked real-time application.

The Entities needed are Tasks (having a private function and workspace), an Interacting Entity we called a Port to synchronize and communicate between the Tasks, see Figure 2. Ports act like channels in the tradition of Hoare's CSP [7], but they allow multiple waiters and asynchronous com-

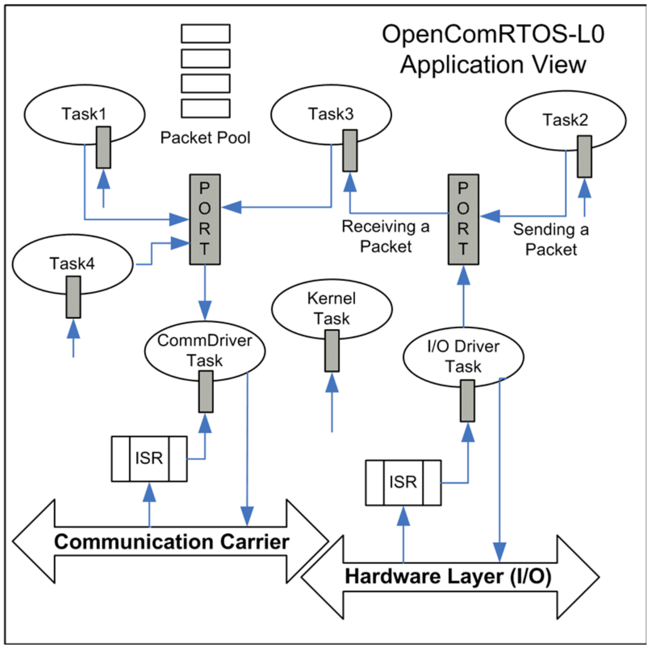

Figure 2. OpenComRTOS-L0 view

munication. One of the Tasks is a kernel Task scheduling the Tasks in order of priority and managing and providing Port based services. Driver Tasks handle inter-node communication. Pre-allocated as well as dynamically allocated Packets are used as carriers for all activities in the RTOS, such as: service requests to the kernel, Port synchronization, data-communication, etc. Each Packet has a fixed size header and data payload with a user defined but global data size. This significantly simplifies the Packet management, particularly at the communication layer. A router function also transparently forwards Packets in order of priority between the network nodes.

In the next semantic level (L1) services and Entities were added, similar to those which can be found in most RTOSs: Boolean events, counting semaphores, FIFO queues, resources, memory pools, etc. The formal modeling leads to the definition of all such Entities as semantic variants of a common and generic entity type. We called this generic entity a “Hub”. In addition, the formal modeling also helped to define “clean” semantics for such services, whereas adhoc implementations often have side-effects. In Table 1 we summarise the semantics.

The services are offered in a non-blocking variant (_NW), a blocking variant (_W), a blocking with timeout variant (_WT) and an asynchronous variant when this makes sense. All services are topology transparent and the mapping of Task and kernel Entities onto this network. See Tables 1 and 2 for details on the semantics.

Using of a single generic entity leads to a much greater code reuse, the resulting code size is at least 10 times less than for an RTOS with a more traditional architecture. One could of course remove all such application-oriented services and just use Hub based services. Unfortunately, this has the drawback that services loose their specific semantic richness, e.g. resource locking clearly expresses that the Task enters a critical section in competition with other Tasks. Also erroneous runtime conditions, like raising an event twice (with loss of the previous event), are easier to detect at application level compared with the case when only a generic Hub is used.

During the formal modeling we also discovered weaknesses in the traditional way priority inheritance is implemented in most RTOSs, fortunately we found a way to reduce the total blocking time. In single processor RTOS systems this is less of an issue, but in multi-processor systems, all nodes can originate service requests and resource locking is a distributed service. Hence, the waiting lists can grow longer and lower priority Tasks can block higher pri-

L1 Entity	Semantics
Event	Synchronisation on a Boolean value.
Counting Semaphore	Synchronisation with counter allowing asynchronous signaling.
Port	Synchronisation with exchange of a Packet.
FIFO queue	Buffered communication of Packets. Synchronisation when queue is full or empty.
Resource	Event used to create a logical critical section. Resources have an owner Task when locked.
Memory Pool	Linked list of memory blocks protected with a resource.

Table 1. Semantics of L1 Entities

Services variants	Synchronising Behavior
“Single-phase” services	
_NW	Non Waiting: when the matching filter fails the Task returns with a RC_Failed.
_W	Waiting: when the matching filter fails the Task waits until such events happens.
_WT	Waiting with a time-out. Waiting is limited in time defined by the time-out value.
“Two-phase” services	
_Async	Asynchronous: when the entity is compatible with it, the Task continues independently of success or failure and will resynchronize later on. This class of services is called “two-phase” services.

Table 2. Service synchronization variant

ority ones while waiting for the resource. This was solved by postponing the resource assignment until the rescheduling moment. Finally, by generalization, also memory allocation has been approached like a resource locking service. In combination with the Packet Pool, this opens new possibilities for safe and secure memory management, e.g. the OpenComRTOS architecture is free from buffer overflow by design.

For the third semantic layer (L2), we will add dynamic support like mobility of code and of kernel Entities. A potential candidate is a light weight virtual machine supporting capabilities as modeled in pi-calculus [11]. This is the subject of further investigations and will be reported in subsequent papers.

2.5. Inherent safety support

By its architecture the L0 and L1 semantic layers are all statically linked, hence an application specific image will be generated by the compiler tools. As we don't consider security risks for the moment, our concern is limited to verifying if the code is inherently safe.

A first level of safety is provided by the formal modeling approach. Each service is intensely modeled and verified with most "corner cases" detected during design time prior to writing code. A second level is provided by the kernel services. All services have well defined semantics. Even when they are asynchronously used, the services become synchronous when resources become depleted. At such moments, a Task is forced to wait allowing other Tasks to proceed and free up resources (like Packets, space in the buffers, etc.). Hence, the systems becomes "self-throttling". A third level is provided by the data structures, mostly based on Packets. All single-phase services use statically allocated Packets which are part of the Task context. These Packets are used for service requests, even when going across processor boundaries. They also carry return values. For two phase services Packets must be allocated from a Packet Pool. When the Pool is empty, the system will start to throttle until Packets are released. Another specific architectural feature is the fact that buffers cannot overflow. In the worst case, the application programmer defined insufficient Packets in the Pool and the buffers will stop growing when all Packets are in use. A last level is the programming environment. All Entities (at L0 and L1) are defined statically, so they are generated together with all other system level data structures by a tool, hence no Entities can be created at runtime. Of course, dynamic support at L2 will require extra support. However, this can only be achieved reliably with hardware support, e.g. to provide protected memory spaces. The same applies to using stack spaces. In OpenComRTOS interrupts are handled on a private and separate stack, so that the Task's stack spaces are not affected. On the MLX16 such a space can be protected, but it is clear that such an inexpensive mechanism should be a must for all embedded processors. A full MMU is not only too complex and too large it is also simply not needed. The kernel has various threshold detectors and provides support for profiling, but the details are outside the scope of this paper.

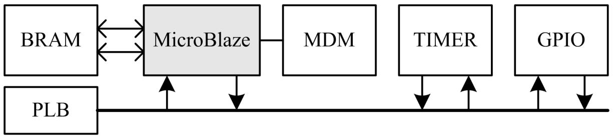

Figure 3. Hardware setup of the test system

3 OpenComRTOS on a MicroBlaze

Field Programmable Gate Arrays (FPGAs) are emerging as an interesting design alternative for system prototyping and implementation for critical applications when the production volume is low [12]. Therefore, we realised the target architecture with the Xilinx Embedded Developer Kit 9.2 and synthesized with Xilinx ISE version 9.2 on a Spartan xc3s1500 Speed Grade -5 FPGA clocked at 50 MHz. Our architecture, shown in Figure 3, is composed of one MicroBlaze processor connected to a Processor Local Bus (PLB). The PLB enables accessing TIMER and GPIO. The TIMER is used to measure the time it takes to execute context switches. The GPIO was used for basic debugging. The processor uses local memory to store code and data of the Task it runs. This memory is implemented through Block RAMs (BRAMs). The MicroBlaze Debug Module (MDM) enables remote debugging of the MicroBlaze processor.

3.1 Code size figures

This section reports the code size figures of OpenComRTOS on the MicrBlaze target. To put these figures into perspective we did two things. First the OpenComRTOS code size figures on the MicroBlaze target are compared with the ones on the MLX16 target. The second comparison is concerned with the code size figures for a simple semaphore example. This example has been implemented using a) using Xilinx Micro-Kernel (XMK) b) OpenComRTOS. The later example is more important, because we can show that OpenComRTOS uses half the memory, compared to the XMK, for the same functionality.

Table 3 reports the code size figures for individual L1 services on MLX16 and MicroBlaze. The total code size of L1 services is just the sum of the individual code sizes. The Grand Total is the sum of all (Total) L1 services and all L0 serves as well as the boot loader, if present. In general

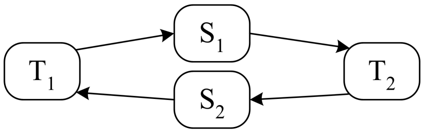

Figure 4. Example project

the code size figures are lower for the MLX16. This is a result of the different processor architectures, MLX16 is a 16bit CISC like microcontroller and MicroBlaze is a 32bit RISC processor. This difference is especially prominent in the Grand Total figures, where the MLX16 requires only about one-third of the code size compared with the MicroBlaze. One reason for this difference is the fact that MLX16 has only 4 registers, which need saving during L0 context switches, compared with 31 for the MicroBlaze.

Service	MLX16	MicroBlaze
L1 Hub shared	400	668
L1 Port	4	8
L1 Event	70	88
L1 Semaphore	54	92
L1 Resource	104	96
L1 FIFO	232	356
Total L1 services	1048	1308
Grand Total	2104	5500

Table 3. OpenComRTOS L1 code size figures MLX16 vs. MicroBlaze

A complete comparison of code size figures between XMK and OpenComRTOS is not possible, because these operating systems offer different services. However, to give an indication of the code size efficiency we implemented a simple application based on two services both OS offer. Figure 4 shows two tasks which exchange messages and synchronise on two semaphores, in both cases 1KiB stack was used. Table 4 shows that OpenComRTOS takes up about one third of the memory used by XMK. This is an important result, because with OpenComRTOS there is more RAM for user applications. This is particularly important when either for speed reasons or for PCB size constraints the complete application has to run in internal (BRAM) memory. One last point in favor of OpenComRTOS is the fact that the kernel can be stripped down to fit in even less memory, e.g. 1KiB L0 with about 300Bytes of data on an MLX16.

4. Conclusions

The OpenComRTOS project has shown that even for software domains often associated with ‘black art’ programming, formal modeling works very well. The resulting software is not only very robust and maintainable but also very performing in size and timings and inherently safer than standard implementation architectures. Its use however must be integrated with a global systems engineering approach as the process of incremental development and modeling is as important as using the formal model checker itself. The use of formal modeling has resulted in many improvements of the RTOS properties.

OS	.text	.data	.bss	total
XMK	12496	348	7304	20148
OpenComRTOS	6016	1008	6320	13344

Table 4. XMK vs OpenComRTOS code size

References

[1] The International Council on Systems Engineering (INCOSE) aims to advance the state of the art and practice of systems engineering. www.incose.org.

[2] The Open License Sociaty researches and develops a systematic systems engineering methodology based on interacting entities and thrustworthy components. www.openlicensesociety.org.

[3] The MISRA Guidelines provide important advice to the automotive industry for the creation and application of safe, reliable software within vehicles. http://www.misra.org.

[4] E. Alkassar, M. A. Hillebrand, D. Leinenbach, N. W. Schirmer, and A. Starostin. The Verisoft Approach to Systems Verification. In *VSTTE 2008*, Toronto, Canada, 2008.

[5] Formal Systems (Europe) Ltd. *Failures-Divergence Refinement: FDR Manual*. http://www.fsel.com/fdr2_manual.html.

[6] A. Gerstlauer, H. Yu, and D. D. Gajski. RTOS Modeling for System Level Design. In *DATE03*, page 10130, Washington, DC, USA, 2003. IEEE Computer Society.

[7] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

[8] G. J. Holzmann. *The SPIN Model Checker : Primer and Reference Manual*. Addison-Wesley Professional, September 2003.

[9] L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[10] M. D. May, P. W. Thompson, and P. H. Welch, editors. *Networks, Routers and Transputers: Function, Performance and Applications*. IOS Press, Amsterdam Netherlands, 1993.

[11] R. Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, June 1999.

[12] A. Tumeo, M. Branca, L. Camerini, M. Ceriani, M. Monchiero, G. Palermo, F. Ferrandi, and D. Sciuto. A dual-priority real-time multiprocessor system on fpga for automotive applications. In *DATE08*, pages 1039–1044, 2008.